Scheduling non-identical jobs on a batch resource

by
Sebastian Kosch

Supervisor: Prof. J. Christopher Beck
April 2013

Scheduling non-identical jobs on a batch resource

Sebastian Kosch

A thesis submitted in conformity with the requirements
for the degree of *Bachelor of Applied Science*

Supervisor: Prof. J. Christopher Beck, MIE

Division of Engineering Science
University of Toronto
April 2013

Contents

List of Figures

List of Tables

Introduction

This paper discusses three different approaches, and several variations on them, to solving the problem of scheduling non-identical jobs on a batch processing machine. Batch processing machines, for the purposes of this paper, can process multiple, non-identical jobs simultaneously—but all jobs must be loaded into and unloaded from the machine at once, which introduces a considerable twist on the "simple parallel resources" known from typical example problems in existing literature.

The machines in question represents real-life resources like autoclaves or ovens, which can process multiple items at a time, but often cannot be opened at random—in fact, such machines often need to wait for the largest item in the batch to be done before the next batch can be inserted.

Malapert et al. [2012] proposed a global constraint programming algorithm consisting of a set of filtering rules to solve the problem. He achieved considerably better speeds than with a simple mixed-integer model, but it seems plausible that the new global constraint is unnecessarily cumbersome to achieve this performance; simple MIP, CP or decomposition approaches are easier to implement and extend. In this paper, we present 1) an improvement to his MIP model, 2) a CP model and 3) a decomposition approach to "divide and conquer" the problem.

1.1 Problem definition

We describe the problem as follows: assume we are given a set of jobs J, each of which has a processing time (or "length") p_j and a size (or "capacity requirement") s_j. Each job also has a due date d_j. The machine is characterized by its capacity b, and in every batch, the jobs' summed sizes must not exceed this number. All values are integer.

The machine can only process one batch k of jobs at a time, and batches always take as long as the longest job in the batch (i.e. $P_k = \max_{j \in k}(p_j)$). Our objective is to minimize the lateness L of the latest job in J, where L is the difference between the job's completion time C_j and its due date d_j—in formal terms, *min.* $L_{\max} = \max_j(C_j - d_j)$. The job's completion time, however, is the completion time of the batch, which in turn finishes with its *longest* job as stated above.

Malapert uses the standard format established by Graham et al. to summarize the problem as $1|p\text{-}batch; b < n; non\text{-}identical|L_{\max}$, where $p\text{-}batch; b < n$ represents the parallel-batch nature and the finite capacity of the resource. A simpler version with identical job sizes was shown to be strongly NP-hard in [Brucker et al., 1998]; this problem, then, is no less difficult.

It helps to visualize the jobs before delving into the technicalities of scheduling them. Figure 1.1 shows a solution to a sample problem with eight jobs and a resource with capacity $b = 20$.

1.2 Organization of this paper

After reviewing some of the most relevant publications on both general MIP/CP models and batch scheduling problems, we first describe Malapert's original MIP model in section 3.1. We then present possible improvements to the model in 3.2. Section 3.3 introduces a CP formulation of the same problem. Sections 3.4.1 and 3.4.1 describe a decomposition approach.

An empirical comparison of the new models and a discussion of the results follow in

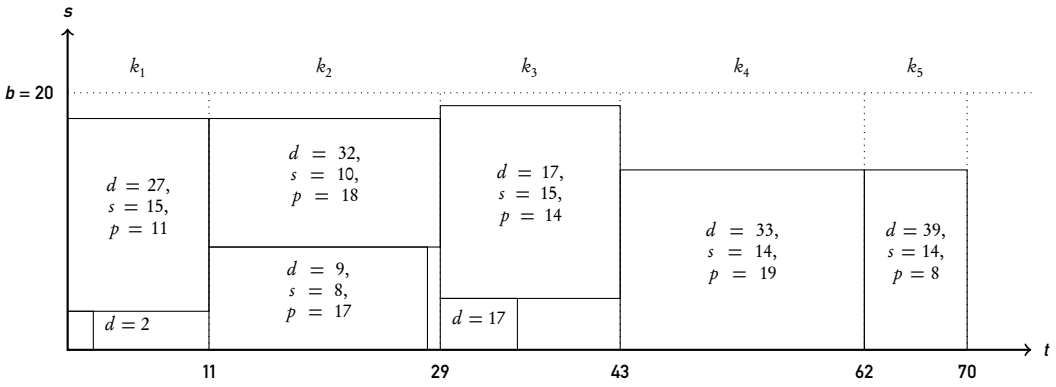


Figure 1.1: Optimal solution to an example problem with eight jobs (s_j and p_j not shown for the two small jobs)

sections 4.1 and 5. Ideas for future work are listed in 6.

Background

Many optimization problems, and scheduling problems in particular, are combinatorial in nature. The number of possible solutions grows exponentially with the number of input variables, and even with fast computers it is impossible to explore all of them individually to find the best one (also called "full enumeration", or "brute-force" search) in a reasonable amount of time. Often, however, it is possible to reason about subsets of solutions that are known to be suboptimal a priori. This limits the search space, allowing us to solve many instances of difficult combinatorial problems in few hours, minutes or even seconds.

Such constrained searches are often implemented in either of two ways (or variants of them): as a *Constraint Programming model* (CP) or as a *Mixed Integer Programming model* (MIP). In this section I will briefly introduce the concepts behind both CP and MIP and review recent work on problems similar to the one dealt with here.

2.1 Constraint Programming

Many problems can be understood as a situation in which a set of values is to be chosen according to certain rules, but since the number of possible combinations of values is enormous, attempting to test all of them is infeasible.

Constraint programming (CP) is a formal framework for the formulation and solution of such problems. CP models consist of a set of variables $V = \{x_1, \ldots, x_n\}$, each x_i of which has a domain $\mathcal{D}_i$, the set of values that could conceivably be assigned to it. The CP problem is defined by a set of relationships that must hold between the variables. Model

2.1.1 illustrates the concept.

$$x \in \{1, \ldots, 10\} \tag{2.1}$$
$$y \in \{9, \ldots, 20\} \tag{2.2}$$
$$z \in \{1, \ldots, 50\} \tag{2.3}$$
$$x > y \tag{2.4}$$
$$z \geqslant xy - 42 \tag{2.5}$$

Model 2.1.1: A simple CP model

Here, x, y, z are the variables with their respective domains.

2.1.1 Propagation

We can choose a variable and a value to assign to it, and explore the consequences of this assignment. Assign $x = 5$, then based on the constraint $x > y$ only values < 5 are permissible for y, but none are available in $\mathcal{D}_y$. The assignment $x = 5$ leads to an *inconsistency*, and another value must be chosen.

If we choose $x = 10$ (which, clearly, is the only feasible value for x, since all others violate $x > y$), y is limited to $y = 9$. This means that $z \geqslant 9 \cdot 10 - 42 = 48$, thus $z = \{48, 49, 50\}$.

This process of successive elimination is known as *propagation*, and it is the core of all CP solver algorithms. In propagation, a variable is assigned a fixed value, and the constraints are used to reduce the domains of other variables accordingly, until no constraints are violated.

2.1.2 Consistency

Propagation is commonly seen as a way to enforce consistency. It can be triggered by fixing a variable to a value, but also by systematic evaluation of constraints. Let any two variables in a problem be involved in a binary constraint C, such as $x > y$ above. If their domains are reduced such that they fulfill C (that is, $\forall a \exists b$ such that $a \in \mathcal{D}_x, b \in \mathcal{D}_y, C(x = a, y =$

$b) = \texttt{true}$), they are *arc-consistent* with regards to C. Propagation will commonly enforce arc consistency across all variables and constraints.

Arc consistency alone is not sufficient to guarantee the existence of a solution (satisfiability), unless the graph of variables and constraints is acyclic. Consider the following problem (Model 2.1.2):

$$x_1 = x_2 \tag{2.6}$$
$$x_1 = x_3 \tag{2.7}$$
$$x_2 = x_4 \tag{2.8}$$
$$x_3 \neq x_4 \tag{2.9}$$
$$x_1, x_2, x_3, x_4 \in \{1, 2\} \tag{2.10}$$

Model 2.1.2: Unsatisfiable CP problem

The model is arc-consistent but has no solution: forcing any variable to assume a fixed value will result in at least constraint being violated.

The notion of arc consistency (two variables) can be extended, e.g. to three variables related through binary constraints (*path consistency*) or to constraints involving more than two variables, where every value assigned to a variable x must be consistent with all of all other variables' values (*generalized arc consistency*).

2.1.3 Global constraints

Propagation makes use of graph-theory based algorithms that enforce a form of consistency on the variables involved in a constraint. Most constraints are formulated in the form of binary relationships involving a single operator (e.g. $\geqslant, =$ or $\neq$). But it is often possible and beneficial to impose constraints on groups of variables at once. Consider the following classic example: This model prescribes that the variables x_1, x_2, x_3, x_4 are to take on different values. The model is arc consistent. To find a solution (or insatisfiability), a value has to be assigned to three of the variables, and the solver has to propagate the domain reductions after every assignment. Consider, on the other hand, the use of a specialized

$$x_1 \neq x_2 \tag{2.11}$$
$$x_2 \neq x_3 \tag{2.12}$$
$$x_3 \neq x_4 \tag{2.13}$$
$$x_4 \neq x_1 \tag{2.14}$$
$$x_1 \neq x_3 \tag{2.15}$$
$$x_2 \neq x_4 \tag{2.16}$$
$$x_1, x_2, x_3, x_4 \in \{1, 2, 3\} \tag{2.17}$$

Model 2.1.3: A clique of not-equal constraints

constraint called `alldifferent`(x_1, x_2, x_3, x_4). This constraint, implemented as a custom algorithm, can speed up the propagation rapidly. In the case of our example, the constraint can infer that the number of variables exceeds the number of available values. The solver can conclude that the problem is insatisfiable without any propagation.

Such specialized or "global" constraints can efficiently exploit properties that hold for certain relationships among groups of variables, and that would not be available if the relationship were expressed as a set of binary constraints.

2.2 Mixed Integer Programming

2.2.1 Linear Programming

Mixed Integer Programs (or "MIP models") express the minimization of a linear function subject to linear constraints. If all variables in the problem can be rational in the solution, the MIP model is really a *linear program* (LP), which can be solved in polynomial time.[1]

Figure 2.1 illustrates the concept of an LP in two variables, x_1 and x_2, as listed in model 2.2.1. The set of feasible solutions is given by the shaded area bounded by the axes and by three inequalities ("constraints"). A third linear term, the objective function $3x_1 + x_2$, is to

[1] In practice, variations on Dantzig's *simplex method* are most often used to solve LPs. Such solvers perform very well on most problems, but no known variant has been proven to have polynomial worst-case complexity [**?**]. Solvers with theoretically polynomial-time complexity exist (Karmarkar's algorithm **?** has a runtime of $\mathcal{O}(n^{3.5}L^2 \cdot \log L \cdot \log\log L)$, for instance, where L is the number of bits of input), but are used less frequently.

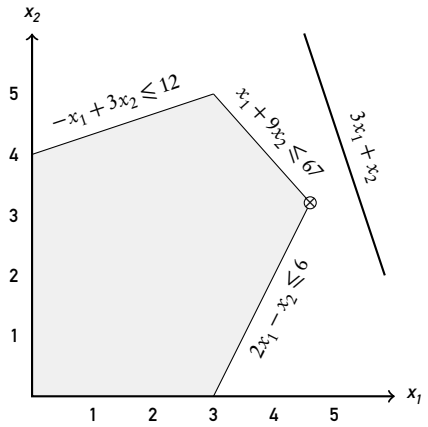


Figure 2.1: Graphical representation of a typical LP problem

be maximized.[2]

$$\begin{aligned}
\text{Maximize} \quad & 3x_1 + x_2 && (2.18)\\
\text{subject to the constraints} \quad & -x_1 + 3x_2 \leqslant 12 && (2.19)\\
& x_1 + 9x_2 \leqslant 67 && (2.20)\\
& 2x_1 - x_2 \leqslant 6 && (2.21)\\
& x_1 \geqslant 0 && (2.22)\\
& x_2 \geqslant 0 && (2.23)
\end{aligned}$$

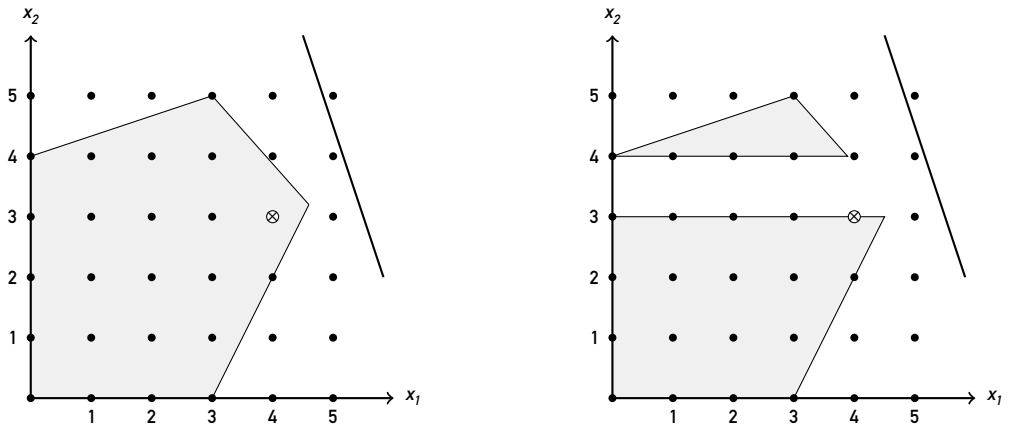
Model 2.2.1: A simple LP model, as shown in figure 2.1

Although the objective function is shown in the figure as a line in a specific location, note that, as it is not an equation, it can be represented by any line parallel to that shown in the figure. While we are trying to maximize the objective value, the solution must be feasible. It is thus obvious that the desired extreme value of our objective function is found at one of the "corners" of the shaded area: the solution is marked ⊗ in the figure. In fact, since the shaded area generated by linear inequalities will always be a convex polygon (or polyhedron, in higher dimensions), the solution will invariably be found at an intersection of hyperplanes.

[2]Minimization is more common, but note that multiplying the objective by -1 achieves this.

2.2.2 Solving MIP models using branch-and-bound

MIP models are LPs in which some of the decision variables are declared integers—thus their name. Like LP models, MIP models also require an objection function to be minimized. Figure 2.2a illustrates this based on the LP problem above: now, only the black dots



(a) at the root node (b) before first branching

Figure 2.2: Graphical representation of a typical MIP problem

represent feasible solutions. While the solution is easily found in the figure by inspection (simply round to the nearest feasible integral solution!), this is not the case in problems with many variables; it is difficult enough to visualize the problem in three dimensions, and many problems require hundreds or thousands. Moreover, rounding is particularly unreliable with variables of small domains (e.g. binary variables), which are often used in MIP models to represent decisions. Indeed, solving MIP models is NP-hard.

Similar to CP searches, MIP searches can be thought of as trees, where every branch represents an assignment of a value to a domain and every leaf represents a feasible solution. The difference lies in the way MIP explores this search tree.

The MIP solver first solves the problem as an LP, which will usually result in fractional values for all or most of the variables. In this context, the LP is known as the *LP relaxation* of the problem—an easier, approximate version of the original. Assuming we are minimizing the objective, the resulting LP objective value will be a lower bound on the optimal

MIP objective value. At this point, the solution is $x_1 = 4.6, x_2 = 3.2$.

The solver then chooses a variable, based on heuristics, and *branches* on it. In this case, assume that we branch on x_2, which means that we set $x_2 \leqslant 3.0 \vee x_2 \leqslant 4.0$. We now have two new MIPs, as shown in figure 2.2b. Both of these new subproblems are need solving.[3] At this point, another heuristic decides which subproblem is solved first. Again, the chosen subproblem is first solved in its LP relaxation, and the above procedure is repeated until an integral solution is found. If this solution is the best solution known so far, it is stored as the *incumbent*. The solver then *backtracks*: it undoes some of the previously fixed variables, and explores other (previously unsolved) subproblems.

This behaviour can be visualized as a search tree in which every node represents a decision between two subproblems. At every node, a subproblem is chosen, the subproblem's LP relaxation is run, and the next variable is chosen to be branched on.

When the solver backtracks to a previous node to explore another subproblem, say P_1, it can compare the LP solution of P_1 with the current incumbent's solution. Since a solution will never improve with additional integrality constraints, the subtree of P_1 can be ignored if the objective value of the LP solution to P_1 is worse than that of the current incumbent. This "pruning" of the search tree during the search is referred to as *bounding*, and is what allows many MIP models to be solved in less time than their theoretically exponential complexity suggests.

2.2.3 Branch-and-cut methods

2.2.4 Special ordered sets

2.2.5 Lazy constraints in MIP

Achieving fast search times in MIP requires striking a delicate balance between keeping the model small (reducing the time needed to solve the LP relaxations) and formulating the model tightly (pruning more branches off the search tree). Often, the way certain

[3] In this example, the optimal value for x_2 was within ± 1 of its LP value. This is often the case, and it is not immediately obvious from the figure why fixing $x_2 = 3.0 \vee x_2 = 4.0$ is insufficient. In some problems, however, the shaded polyhedron protudes relatively far diagonally through the grid of integer solutions, in which case branching on $\leqslant \lfloor x_i \rfloor$ and $\geqslant \lceil x_i \rceil$ is necessary to capture the optimal feasible solution.

constraints can be used to perform pre-solve reductions plays into the performance also, and is difficult to predict.

Lazy constraints are one way to avoid this problem. They are expressed as regular linear constraints and checked against whenever the model encounters a new solution. Lazy constraints that were violated are then added to the model to be used in future instances.

This is particularly useful when a problem lends itself to the generation of vast sets of constraints involving few variables, but most of these constraints are likely not needed. In such cases, lazy constraints can greatly speed up solution times.

2.3 Scheduling using CP and MIP

2.3.1 Common types of scheduling problems

2.3.2 Temporal problems in CP

2.3.3 Temporal problems in MIP

2.4 Literature

The problem at hand is based on the work of Malapert et al. [2012], who proposed a global constraint `sequenceEDD` to be used in combination with `pack` to solve it to optimality. The `sequenceEDD` constraint is implemented as four distinct filtering rules applied at relevant domain changes: three to update bounds on $L_{\max}$ based on different conditions, and one to limit the number of batches based on the marginal cost difference between adding a job to an empty batch vs. an existing one. The new global was shown to significantly outperform a simple MIP model of the same problem.

Other authors have examined similar problems: Azizoglu and Webster [2000] provide an exact method and a heuristic for the same problem, but minimize makespan ($C_{\max}$) instead of $L_{\max}$, as have Dupont and Dhaenens-Flipo [2002]. Similar exact methods have been proposed for multi-agent variants with different objective functions [Sabouni and Jolai, 2010], for makespan minimization on single batch machines [Kashan et al., 2009], and

for makespan minimization on parallel batch machines with different release dates [Ozturk et al., 2012]. A more extensive review of MIP model applications in batch processing is given by Grossmann [1992].

Modelling the problem

In this section we present the models examined. Beginning with the MIP model used by Malapert, we explore several additional constraints that, while redundant, tighten the search space. We comment on the performance of the approaches (see section **??** for a comprehensive comparison of performance). We present a CP model and a decomposition-based approach using either MIP or CP. Finally, we introduce a new approach to setting up a MIP model that significantly improves upon the original MIP and outperforms all other models presented in this paper.

3.1 MIP model

As a baseline to compare other models' performance to, we first replicated Malapert's original MIP approach, as given in Model 6.5.1. It uses a set of binary decision variables x_{jk} to represent whether job j is assigned to batch k. The original model assumes a set K of $|K| = |J|$ batches, the number of jobs being a trivial upper bound on the number of batches required; it also enforces an earliest-due-date-first (EDD) ordering of the batches (constraint 3.7).

3.2 Improved MIP model

Several improvements can be made to Malapert's MIP model in the form of additional constraints shown in Model 3.2.1, as described in greater detail in the subsections below.

$$
\begin{aligned}
\text{Min.} \quad & L_{\max} && && (3.1)\\
\text{s.t.} \quad & \sum_{k\in K} x_{jk} = 1 && \forall j \in J && (3.2)\\
& \sum_{j\in J} s_j x_{jk} \leqslant b && \forall k \in K && (3.3)\\
& p_j x_{jk} \leqslant P_k && \forall j \in J, \forall k \in K && (3.4)\\
& C_{k-1} + P_k = C_k && \forall k \in K && (3.5)\\
& (d_{max} - d_j)(1 - x_{jk}) + d_j \geqslant D_k && \forall j \in J, \forall k \in K && (3.6)\\
& D_{k-1} \leqslant D_k && \forall k \in K && (3.7)\\
& C_k - D_k \leqslant L_{\max} && \forall k \in K && (3.8)\\
& C_k \geqslant 0, P_k \geqslant 0 \text{ and } D_k \geqslant 0 && \forall k \in K && (3.9)
\end{aligned}
$$

Model 3.1.1: Malapert's original MIP model

3.2.1 Grouping empty batches

The given formulation lacks a rule that ensures that no empty batch is followed by a non-empty batch. Empty batches have no processing time and a due date only bounded by $d_{\max}$, so they can be sequenced between non-empty batches without negatively affecting $L_{\max}$. Since, however, desirable schedules have no empty batches scattered throughout, we can easily reduce the search space by disallowing such arrangements. The idea is illustrated in Figure 3.1.

A mathematical formulation is

$$\sum_{j\in J} x_{j,k-1} = 0 \rightarrow \sum_{j\in J} x_{jk} = 0 \quad \forall k \in K. \tag{3.16}$$

To implement this, we can write constraints in terms of an additional set of binary variables, e_k, indicating whether a batch k is empty or not:

$$\sum_{j\in J} x_{j,k-1} = 0 \rightarrow \sum_{j\in J} x_{jk} = 0 \quad \forall k \in K \tag{3.10}$$

$$e_k + \sum_{j\in J} x_{jk} \geqslant 1 \quad \forall k \in K \tag{3.11}$$

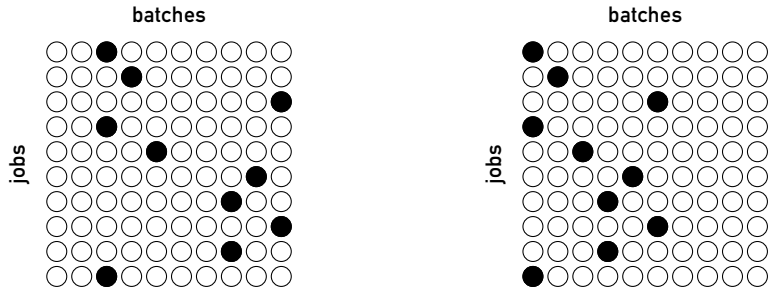
$$n_j(e_k - 1) + \sum_{j\in J} x_{jk} \leqslant 0 \quad \forall k \in K \tag{3.12}$$

$$e_k - e_{k-1} \geqslant 0 \quad \forall k \in K \tag{3.13}$$

$$x_{jk} = 0 \quad \forall \{j \in J, k \in K | j > k\} \tag{3.14}$$

$$L_{\max} \geqslant \lceil \frac{1}{b} \sum_j s_j p_j \rceil - \delta_q \quad \forall q, \forall \{j \in J | d_j \leqslant \delta_q\} \tag{3.15}$$

Model 3.2.1: Improvements to Malapert's original MIP model



(a) Without dominance rule (b) With dominance rule

Figure 3.1: Dominance rule to eliminate empty batches followed by non-empty batches (circles represent the x_{jk} variables; a filled circle stands for $x_{jk} = 1$)

$$e_k + \sum_{j\in J} x_{jk} \geqslant 1 \quad \forall k \in K, \tag{3.17}$$

$$n_j(e_k - 1) + \sum_{j\in J} x_{jk} \leqslant 0 \quad \forall k \in K. \tag{3.18}$$

Constraints (3.17) enforce $e_k = 1$ when the batch k is empty. Constraints (3.18) enforce $e_k = 0$ otherwise, since the sum term will never exceed n_j. The rule (3.16) can now be expressed as $e_{k-1} = 1 \rightarrow e_k = 1$, and implemented as follows:

$$e_k - e_{k-1} \geqslant 0 \quad \forall k \in K. \tag{3.19}$$

We can also prune any attempts to leave the first batch empty by adding a constraint $e_0 = 0$.

3.2.2 No postponing of jobs to later batches

Since the jobs are already sorted by non-decreasing due dates, it makes sense to explicitly instruct the solver never to attempt to push jobs into batches with a greater index than their own: even if every job had its own batch, it would be unreasonable to ever postpone a job to a later batch.

$$x_{jk} = 0 \quad \forall \{j \in J, k \in K | j > k\} \tag{3.20}$$

3.2.3 Lower bound on $L_{\max}$

Let a *bucket* q denote the set of all batches with due date δ_q. Then the completion date C_q of this bucket is the completion date of the last-scheduled batch with due date δ_q, and the lateness of the bucket q is $L_q = C_q - \delta_q$. Since all batches up to and including those in bucket q are guaranteed to contain all jobs with due dates $d \leqslant \delta_q$ – as ensured by the EDD ordering of batches – the lower bound on every bucket's lateness $LB(L_q)$ is a valid lower bound on $L_{\max}$. In other words, jobs with due date $d \leqslant \delta_q$ will be found only in batches up to and including the last batch of bucket q. This provides a lower bound on the lateness of bucket q:

$$L_{\max} \geqslant C_{\max,q} - \delta_q \quad \forall q \tag{3.21}$$

The buckets up to bucket q will likely also contain some later ($d > \delta_q$) jobs in the optimal solution but this does not affect the validity of the lower bound.

Now we need to find $C_{\max,q}$, or at least a lower bound on it, in polynomial time. The simplest approach simply considers the jobs' total "area" (or "energy"), i.e. the sum of all

$s_j p_j$ products:

$$C_{\max,q} \geqslant \left\lceil \frac{1}{b} \sum_j s_j p_j \right\rceil \quad \forall q, \forall \{j \in J | d_j \leqslant \delta_q\} \tag{3.22}$$

A better lower bound on $C_{\max,q}$ would be given by a preemptive-cumulative schedule. Unfortunately, minimizing $C_{\max}$ for such problems is equivalent to solving a standard bin-packing problem, which requires exponential time.[1]

3.3 CP model

The constraint programming model is based on a set of decision variables $B_j = \{k_1, \ldots, k_{n_k}\}$, where each variable stands for the batch job j is assigned to. The complete model is given in Model 3.3.1.

$$\begin{aligned}
\text{Min.} \quad & L_{\max} && && (3.23)\\
\text{s.t.} \quad & \texttt{pack}(J, K, b) && && (3.24)\\
& \texttt{cumul}(J, b) && && (3.25)\\
& P_k = \max_j p_j && \forall \{j \in J | B_j = k\}, \forall k \in K && (3.26)\\
& D_k = \min_j d_j && \forall \{j \in J | B_j = k\}, \forall k \in K && (3.27)\\
& C_k + P_{k+1} = C_{k+1} && \forall k \in K && (3.28)\\
& L_{\max} \geqslant \max_k (C_k - D_k) && && (3.29)\\
& \texttt{IfThen}(P_k = 0, P_{k+1} = 0) && \forall k \in \{k_1, \ldots, k_{n_k - 2}\} && (3.30)\\
& B_j \leqslant k && \forall \{j \in J, k \in K | j > k\} && (3.31)
\end{aligned}$$

Model 3.3.1: Constraint programming model

Constraint (3.24) makes sure the jobs are distributed into the batches such that no batch exceeds the capacity b. Constraint (3.25), a global cumulative constraint, keeps jobs from

[1] In a preemptive-cumulative schedule, jobs may be stopped and restarted mid-execution, but occupy a constant amount s_j on the resource while executing. In such a schedule, minimizing the makespan is as difficult as solving a bin-packing problem: we can break jobs into small pieces (no longer than the smallest common divisor of the jobs' lengths p) and then pack them together such as to minimize the number of small time slots needed.

overlapping on the resource – while redundant with (3.24), this speeds up propagation slightly. Constraints (3.26), (3.27), (3.28) and (3.29) define P_k, D_k, C_k (batch completion time) and $L_{\max}$, respectively.

3.3.1 Grouping empty batches

We can force empty batches to the back and thus establish dominance of certain solutions. The implementation is much easier than in the MIP model:

$$\texttt{IfThen}(P_k = 0, P_{k+1} = 0) \quad \forall k \in \{k_1, \ldots, k_{n_k - 1}\} \tag{3.32}$$

3.3.2 No postponing of jobs to later batches

Just like in the MIP model, jobs should never go into a batch with an index greater than their own:

$$B_j \leqslant k \quad \forall \{j \in J, k \in K | j > k\} \tag{3.33}$$

3.4 Decomposition approach

Instead of solving the entire problem using one model, we can solve the problem step by step, using the best techiques available for each subproblem. This approach is inspired by a method called *Benders decomposition*.

3.4.1 Branch-and-bound by batch in chronological order

A basic version of this approach uses branch-and-bound to transverse the search tree. At each node on level ℓ, a single MIP and/or CP model is run to assign jobs to batch $k = \ell$. The remaining jobs are passed to the children nodes, which assign jobs to the next batch, and so on – until a solution, and thus a new upper bound on $L_{\max}$, is found. Several constraints are used to prune parts of the search tree that are known to offer only solutions worse than this upper bound. Figure 3.2 shows an example in which a MIP model is used to assign jobs to the batch at every node. Algorithm 1 outlines what happens at each node:

Algorithm 1 MIP node class code overview

```
update currentAssignments                                  ▷ this keeps track of where we are in the tree
if no jobs given to this node then                         ▷ if this is a leaf node, i.e. all jobs are assigned to batches
    calculate L_max,current based on currentAssignments
    if L_max,current < L_max,incumbent then
        L_max,incumbent ← L_max,current
        bestAssignments ← currentAssignments
    end if
    return to parent node
end if
set up MIP model                                           ▷ as described below
repeat
    x_j ← model.solve(x_j)                                 ▷ let model assign jobs to the batch
    spawn and run child node with all {j|x_j = 0}          ▷ pass unassigned jobs to children
    add constraint to keep this solution from recurring    ▷ this happens once the child node returns
until model has no more solutions
return to parent node
```

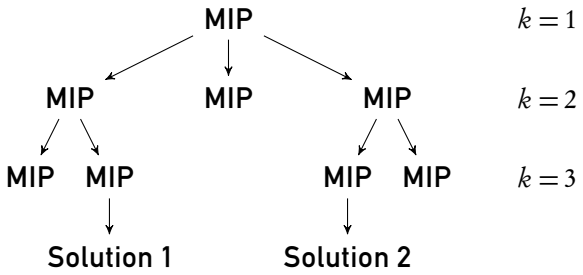


Figure 3.2: Batch-by-batch decomposition using MIP only

the model finds the best jobs to assign to the batch according to some rule, lets the children handle the remaining jobs, and tries the next best solution once the first child has explored its subtree and backtracked.

Using MIP and cumulative packing after the batch

The first version of the branch-and-bound batch-by-batch decomposition uses a MIP model at each node to assign jobs to the respective batch. The remaining jobs are packed such as to minimize their $L_{\max}$, with a relaxation of the batching requirement, i.e., as if on a cumulative resource.

Model 3.4.1 implements a time-indexed cumulative constraint on the non-batched jobs. Constraints (3.35) through (3.37) ensure that the batch stays below capacity, define the

$$\text{Min.} \quad L_{\max,\text{cumul}} \tag{3.34}$$

$$\text{s. t.} \quad \sum_j s_j x_j \leqslant b \qquad \forall j \in J \tag{3.35}$$

$$P_k \geqslant p_j x_j \qquad \forall j \in J \tag{3.36}$$

$$\sum_j x_j \geqslant 1 \qquad \forall \{j \in J | d_j = \min(d_j)\} \tag{3.37}$$

$$P_k + \frac{1}{b}\sum_i s_i p_i \leqslant d_j + L_{\max,\text{incmb}} - 1 - v_k \qquad \forall j \in J, \forall \{i \in J | d_i \leqslant d_j\} \tag{3.38}$$

$$\sum_t u_{jt} = 1 \qquad \forall j \in J \tag{3.39}$$

$$\sum_j \sum_{t' \in T_{jt}} s_j u_{jt'} \leqslant b \qquad \forall t \in \mathcal{H} \tag{3.40}$$

$$(v_k + t + p_j)u_{jt} \leqslant d_j + L_{\max,\text{incmb}} - 1 \qquad \forall j \in J, \forall t \in \mathcal{H} \tag{3.41}$$

$$L_{\max,\text{cumul}} \geqslant (v_k + t + p_j)u_{jt} - d_j \qquad \forall j \in J, \forall t \in \mathcal{H} \tag{3.42}$$

$$u_{j,t=0} = x_j \qquad \forall j \in J \tag{3.43}$$

$$u_{it} \leqslant (1 - x_j) \qquad \forall i, j \in J, \forall t \in \{1, \ldots, p_j - 1\} \tag{3.44}$$

$$b - \sum_{i \in J} s_i x_i \leqslant (b w_j + 1)s_j \qquad \forall j \in J \tag{3.45}$$

$$P_k + 2w_j n_t \geqslant p_j + n_t x_j \qquad \forall j \in J \tag{3.46}$$

$$P_k - 2(1 - w_j)n_t \leqslant p_j + n_t x_j - 1 \qquad \forall j \in J \tag{3.47}$$

Model 3.4.1: MIP model in batch-by-batch branch-and-bound

duration of the batch P_k and force at least one of the earliest-due jobs into the batch.

Constraints (3.38) express the interval relaxation used to ensure that no jobs exceed their latest allowable finish date given any batch assignment. Even jobs that are assigned to the batch have to fulfill this requirement.

Constraints (3.39) and (3.40) implement the cumulative nature of the post-batch assignments by ensuring that each job starts only once, and no jobs overlap on a given resource at any time.

x_j	is 1 iff job j is assigned to the batch
u_{jt}	is 1 iff job j starts in time slot t
T_{jt}	is the set of all time slots occupied by job j if it ended at time t, that is $T_{jt} = \{t - p_j + 1, \ldots, t\}$
v_k	is the start time of the batch at the given node in the search tree
$L_{\text{max,incmb}}$	is the incumbent (known best) value of and thus an upper bound on L_{max}
$\mathcal{H}$	is the set of all indexed time points $\{0, \ldots, n_t - 1\}$
w_j	is 1 iff job j is either longer than the batch ($p_j > P_k$) or already part of the batch ($x_j = 1$). Neither condition must be fulfilled for constraint (3.45) to have an effect on job j

Table 3.1: Notation used in the decomposition model

Constraints (3.41) again limit the possible end dates of a job, but unlike (3.38), they use the time assignments on the cumulative resource to determine end dates. Constraints (3.42) define the value of $L_{\text{max,cumul}}$, the maximum lateness of any job in the non-batched set.

Constraints (3.43) force batched jobs to start at $t = 0$, while (3.44) force *all* jobs to start either at $t = 0$ or after the last batched job ends.

Constraints (3.45) enforce a dominance rule: jobs must be assigned to the batch such that the remaining capacity, $b_r = b - \sum_j s_j x_j$, is less than the size s_j of the *smallest* job from the set of non-batched jobs that are *not longer* than the current batch. That is, if there exists an non-batched job j with $p_j \leqslant P_k$ and $s_j \leqslant b_r$, then the current assignment of jobs is infeasible in the model. The reasoning goes as follows: given any feasible schedule, assume there is a batch k_a with b_r remaining capacity and a later batch k_b containing a job j such that $s_j \leqslant b_r$ and $p_j \leqslant P_{k_a}$. Then job j can always be moved to batch k_a without negatively affecting the quality of the solution: if the schedule was optimal, then moving j will not affect L_{max} at all (otherwise, it was no optimal schedule); if the schedule was not optimal, then L_{max} will stay constant (unless j was the longest job in k_b and L_{max} occured in or in a batch after k_b, in which case L_{max} will be improved).

This rule is implemented by means of a binary variable w_j, which, as defined by constraints (3.45) and (3.46), is 1 iff $p_j > P_k \vee x_j = 1$. These are the cases in which a job j is *not*

to be considered in (3.44), and so w_j is used to scale s_j to a value insignificantly large in the eyes of the constraint's less-than relation.

After a solution is found, a child node in the search tree has run the subtree and returned, a constraint of the form

$$\sum_j x_j + \sum_i (1 - x_i) \leqslant n_j - 1 \quad \forall\{j \in J | x_j = 1\}, \forall\{i \in J | x_i = 0\} \tag{3.48}$$

is added to the model before the solver is called again, to exclude the last solution from the set of feasible solutions.

Using CP and cumulative packing after the batch

This approach is equivalent, but we now use CP to select the batch assignments, again based on a minimized $L_{\max}$ among the non-batched jobs. Model 3.4.2 uses interval variables j to represent the jobs; time constraints on the jobs ("est", "lft" etc.) are represented as functions such as `startOf`(j) and `endOf`(j).

Contraints (3.50) through (3.53) bi-directionally define the relationship between a job's x_j and `startOf`(j): $x_j = 1$ is equivalent with a start time of $t = 0$, and $x_j = 0$ is equivalent with a start time of $t \geqslant P_k$. The term $\sum_{i \in J} p_i$ is used as a large constant as it is greater than any job's start time.

Constraint (3.57) is equivalent to constraints (3.45) through (3.47) in Model 3.4.1 above. The cumulative constraint (3.60) ensures that jobs do not overlap.

3.5 A move-based MIP model

This approach is based on the idea of preserving EDD ordering among the batches.

Initially, all jobs are assigned to a single batch, ordered by non-decreasing due date, and by non-decreasing processing time in case of a tie between two jobs. For the purposes of this paper, we refer to this schedule as the *single-EDD* schedule. We can calculate the lateness $L_{k,\text{single}}$ of every job in this schedule in linear time.

$$\text{Min.} \quad L_{\max} \tag{3.49}$$

$$\text{s.t.} \quad \texttt{startOf}(j) \geqslant (1 - x_j) P_k \qquad \forall j \in J \tag{3.50}$$

$$\texttt{startOf}(j) \leqslant (1 - x_j) \sum_{i \in J} p_i \qquad \forall j \in J \tag{3.51}$$

$$x_j \geqslant \frac{\frac{1}{2} - \texttt{startOf}(j)}{\sum_{i \in J} p_i} \qquad \forall j \in J \tag{3.52}$$

$$x_j \leqslant 1 + \frac{\frac{1}{2} - \texttt{startOf}(j)}{\sum_{i \in J} p_i} \qquad \forall j \in J \tag{3.53}$$

$$P_k \geqslant \max_{j \in J}(x_j p_j) \tag{3.54}$$

$$\texttt{endOf}(j) = d_j + L_{\max,\text{incmb}} - 1 \qquad \forall j \in J \tag{3.55}$$

$$L_{\max} \geqslant \max_{j \in J}(\texttt{endOf}(j) - d_j) \tag{3.56}$$

$$\texttt{IfThen}(p_j \leqslant P_k \wedge x_j = 0, b - \sum_{j \in J} s_j x_j \leqslant s_j) \qquad \forall j \in J \tag{3.57}$$

$$P_k + \frac{1}{b} \sum_i s_i p_i \leqslant d_j + L_{\max,\text{incmb}} - 1 - v_k \qquad \forall j \in J, \forall \{i \in J | d_i \leqslant d_j\} \tag{3.58}$$

$$\sum_j x_j \geqslant 1 \qquad \forall \{j \in J | d_j = \min(d_j)\} \tag{3.59}$$

$$\texttt{cumul}(J, b) \tag{3.60}$$

Model 3.4.2: CP model in batch-by-batch branch-and-bound

For the purpose of this discussion, we use the following terminology: in any batch k holding multiple jobs in an EDD schedule, let *host job* denote the earliest-due job in the batch, and *guest jobs* all other jobs. If a batch k only holds one job j, then job j is said to be *single*.

We now observe the following two things:

Proposition 1. *Consider a schedule in which batches are ordered by EDD. Then moving job j from batch k_β into an earlier batch k_α will preserve EDD ordering if j is single in k_β and if k_α is not empty.*

Proof. Moving j into k_α will leave D_α unaffected, since k_α is not empty, i.e. there is a host

job in k_α. Since the original schedule was EDD-ordered, the host in k_α is due earlier than j.

Batch k_β will be reduced to zero processing time, but batches after k_β will still be due after $k_{\beta-1}$, so EDD ordering is preserved. □

Proposition 2. *Starting from a single-EDD schedule, moving single jobs back into earlier non-empty batches will generate all possible EDD schedules.*

Proof. The order in which moves are performed is arbitrary; moves are independent of each other (capacity requirements notwithstanding). □

Consider a single-EDD schedule, that is, for every job and batch, the indices match: $B_j = j \forall j$ (where B_j denotes the batch j is assigned to). Moving j from batch k_j into an earlier batch k_α has the following effect:

- the lateness of all batches after k_j is reduced by p_j,
- the lateness of all batches after k_α, including those after k_j, is increased by $\max(0, p_j - P_\alpha)$.

Since, in any batch k, only the host job (with index $j = k$) is relevant to $L_{\max}$, always being the earliest-due, we can understand the lateness of batch k as its single-EDD lateness $L_{k,\text{single}}$, modified by summed effect all moves have on it as listed above.

The following expression defines the lateness of a batch k based on this calculation:

$$L_k = L_{k,\text{single}} - \sum_{h=0}^{k} P_h - p_h(2 - x_{hh}) \quad \forall k \in K \tag{3.61}$$

where x_{jk} is 1 iff job j is moved back into batch k. The sum term represents the effects that all moves have on previous batches up to, and including, k: $x_{hh} = 1$ iff job h is not moved from its single-EDD batch h, and so the sum term evaluates to $P_h - p_h$ for a batch in which the host was not moved. If h has guests, then $P_h - p_h$ may be positive; otherwise it will be zero. Effectively, for batch h, the time $\max(0, P_h - p_h)$ is added to L_k. If, however,

job h was moved out of its batch, then the batch will have a processing time of $P_h = p_h$ (the fixed lower bound). Then, effectively, for batch h, the time p_h is subtracted from L_k.

The full set of constraints is listed in Model 3.5.1, and the following section explains every constraint in detail.

Constraints (3.63) implement the requirement that jobs are only moved into earlier batches.

Constraints (3.64) apply to jobs with index greater than f_L, which is the index of the job with the largest value for $L_{k,\text{single}}$. All jobs after batch f_L can be ignored in the question, based on the proof given in Appendix **??**. This constraint fixes those jobs to their single-EDD batches, effectively removing them from the model. This set of constraints is rarely active but can greatly decrease the size of the model in some instances.

Constraints (3.65) define the value of P_k for every batch k as the longest p of all jobs in k. This is required in the definition of batch lateness as described above and in (3.66).

Constraints (3.67) ensure that no job is moved into a host-less batch, i.e. in order to move job j into batch k ($x_{jk} = 1$), job k must still be in batch k ($x_{kk} = 1$).

Constraints (**??**) and (**??**) are capacity constraints and uniqueness constraints: batches have to remain within capacity b, and every job can only occupy one batch.

Constraint (**??**) and (**??**) are dominance rules implemented as lazy constraints as explained in subsection 2.2.5, and are presented in greater length in the following subsections.

Symmetry-breaking rule

Constraint (**??**) implements the following concept: if two jobs j_1 and j_2 are assigned as guests to batches k_1 and k_2, and no other jobs (save for the hosts) are assigned to k_1 and k_2, then j_1 is always assigned to k_1 and j_2 to k_2.

Additionally, both jobs have to be shorter (in terms of p) than the shorter of the two batches, and all possible assignments have to be feasible capacity-wise.

Large problem instances generate thousands of such constraints, and only very few of them will ever hold in the model. Declaring them as "lazy" helps keep the model small.

$$\text{Min.} \quad L_{\max} \tag{3.62}$$

$$\text{s.t.} \quad x_{jk} = 0 \qquad \forall\{j \in J, k \in K | j > k\} \tag{3.63}$$

$$x_{jj} = 1 \qquad \forall\{j \in J | j \geqslant f_L\} \tag{3.64}$$

$$P_k \geqslant p_j x_{jk} \qquad \forall\{j \in J, k \in K | j \geqslant k\} \tag{3.65}$$

$$L_{\max} \geqslant L_{k,\text{single}} - \sum_{h=0}^{k} P_h - p_h(2 - x_{hh}) \qquad \forall k \in K \tag{3.66}$$

$$x_{jk} \leqslant x_{kk} \qquad \forall\{j \in J, k \in K | j > k\} \tag{3.67}$$

$$\sum_j s_j x_{jk} \leqslant b \qquad \forall k \in K \tag{3.68}$$

$$\sum_k x_{jk} = 1 \qquad \forall j \in J \tag{3.69}$$

$$(*) \quad \begin{aligned} &2(4 - x_{k_1,k_1} - x_{k_2,k_2} \\ &+ \sum_{\substack{j \\ j \neq j_1 \\ j \neq j_2}} -x_{j_1,k_1} - x_{j_1,k_2} - x_{j_2,k_1} - x_{j_2,k_2}) \geqslant x_{j_1,k_2} + x_{j_2,k_1} \end{aligned} \qquad \begin{aligned} &\forall\{j_1, j_2 \in J, \\ &k_1, k_2 \in K \\ &| \, k_1 < k_2 < j_1 < j_2 \wedge \\ &[p_i \leqslant k_h \wedge b - s_h \geqslant s_i \\ &\forall i \in \{j_1, j_2\}, \\ &\forall h \in \{k_1, k_2\}]\} \end{aligned} \tag{3.70}$$

$$(*) \quad x_{j_1,k} \geqslant x_{j_2,k} \qquad \begin{aligned} &\forall\{k \in K, j_1 \in J, j_2 \in J \\ &| \, k < j_1 < j_2 \\ &\wedge s_{j_1} \leqslant s_{j_2} \\ &\wedge s_{j_2} + s_k \leqslant b \\ &p_{j_1} \leqslant p_k \wedge p_{j_1} \geqslant p_{j_2}\} \end{aligned} \tag{3.71}$$

Model 3.5.1: Move-based MIP model. Constraints marked $(*)$ are lazy constraints (see subsection 2.2.5 for details).

Dominance rule on "safe" moves

This applies to pairs of jobs $j_1 < j_2$ that are, potentially, competing to be guests in batch k. If both j_1 and j_2 are shorter (in terms of p) than k, we can safely enforce that in some cases, j_1 gets priority over j_2, namely if both $p_{j_1} \geqslant p_{j_2}$ and $s_{j_1} \leqslant s_{j_2}$ hold.

Two facts allow for this set of rules:

Proposition 3. *If a job j_h can either be safely moved into an earlier batch, or act as a host for later jobs $J_g = \{j_{g_1}, j_{g_2}, \ldots\}$, moving j_h is always preferable.*

Proof. Let $k_{L_{\max}}$ be the batch hosting the job with maximum lateness in the final solution. If j_h is due after $k_{L_{\max}}$, moving it will have no impact on $L_{\max}$ (see Appendix **??**). If j_h is due before $k_{L_{\max}}$, but the earliest-due of J_g is due after $k_{L_{\max}}$, then moving J_g into j_h will have no impact or a worsening impact on $L_{\max}$. If both j_h and J_g are due before $k_{L_{\max}}$, then moving J_g will, at best, improve $L_{\max}$ by p_{j_h} (if $\max_p(J_g) \geqslant p_{j_h}$). In this case, however, moving j_h back safely will, under any circumstances, reduce $L_{\max}$ by p_{j_h}. The first job in J_g can host all later jobs in J_g (and possibly more, since vertical capacity was freed). The total reduction in $L_{\max}$ in this case is improved. □

Proposition 4. *If a job j_1 with $d_{j_1} \leqslant d_{j_2}$ is both longer and thinner than j_2 ($p_{j_1} \geqslant p_{j_2}, s_{j_1} \leqslant s_{j_2}$), assigning it as a guest to k will produce a lower L_{max} than assigning j_2 to k.*

Proof. Because of due date and size requirements, j_1 is a potential host for j_2, Proposition 3 applies.

Generally, a longer guest job will result in a greater reduction in $L_{\max}$, and is thus preferable. Since j_1 is thinner than j_2, the dominance rule will not preclude (via capacity constraints) other guest jobs from being assigned to k as a consequence of prioritizing j_1 over j_2. □

Results

4.1 Empirical comparison of described models

The models were tested on a set of job lists. Malapert's paper uses benchmark job lists by Daste et al. [2008a,b]. Since neither publication is available online, I created my own set of randomized job lists for the purposes of this paper, with $s_j, p_j \in [1, 20]$ and $d_j \in [1, 10n_j]$ where n_j is the number of jobs.[1] Ten different sample job sets are used per unique value of n_j. The times shown in figure 4.1 are averaged over those ten instances for each n_j.

The models were run on an i7 Q740 CPU in single-thread mode, with 8 GB RAM. Solving was aborted after a time of 3600 seconds (1 hour).

The CP branch-and-bound model times out on most instances and is not shown here. The CP model times out on one 12-job instance and gets progressively worse with more jobs; similar to Malapert's original MIP model.

The final version of this document will include all results for 40 instances per n_j and for all models in tabulated form, as well as graphs to show how average solution quality (LB/UB gap) improves over time.

[1] These instances will be updated to reflect the feature distributions used in Daste et al.

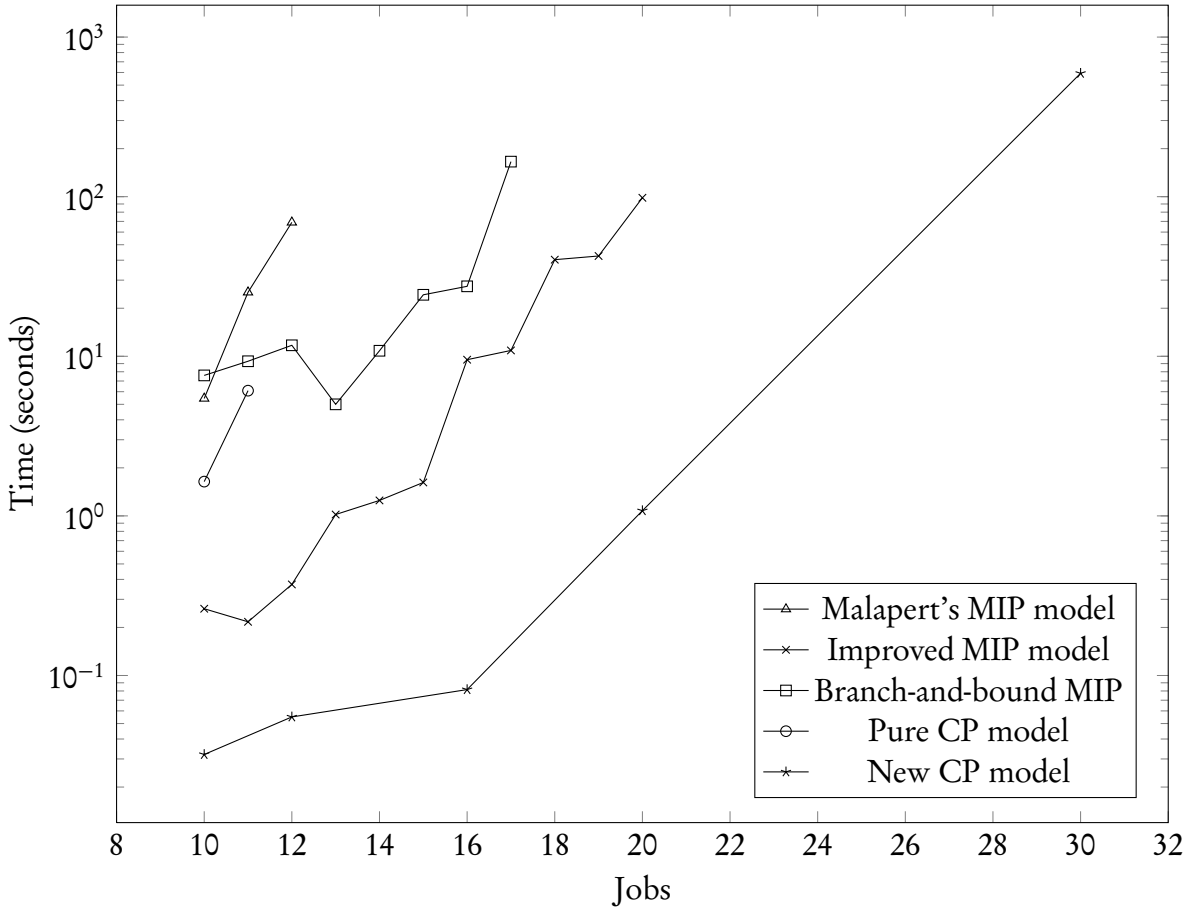


Figure 4.1: Comparison of CPU time used by different models to find an optimal schedule and prove its optimality.

Discussion

The improved MIP model currently performs best, mostly owed to constraint (3.20) which excludes a large number of potential batch assignments.

The batch-by-batch decomposition using MIP to assemble batches, while an improvement over Malapert's MIP model, is not satisfactory; the relaxations are too lax for the decomposition to compensate for the overhead of a naive branch-and-bound search.

The goal is to solve instances with up to 50 jobs in less than an hour (a performance comparable to that of Malapert's global constraint).

Future Work

6.1 Schedule

Current work on a new MIP model (not shown in this document) will continue. I will particularly try to incorporate bounds on $L_{\max}$ and $C_{\max}$ as given in papers such as [Azizoglu and Webster, 2000] and [Dupont and Dhaenens-Flipo, 2002]. Some of the CP model improvements described below shall be implemented as well, and I intend to finish the experiments by late Feburary.

The next step, an extensive discussion of the results, is planned for early March. This will involve researching how different test instances affect solving time in different models. A final draft of the thesis shall be finished by mid-March.

6.2 MIP model

6.2.1 Upper bound on $L_{\max}$

An upper bound on $L_{\max}$ can be found by using a dispatch rule to find a feasible, if not optimal, schedule. A good approach could be the "best-fit" heuristic proposed in Malapert's paper.

6.2.2 Bounding the number of batches n_k

Initially, the number of batches needed is assumed to be equal to the number of jobs: $n_k = n_j$. Reducing n_k by pre-computing the maximum number of batches needed shrinks

the x_{jk} matrix, and prunes potential search branches in branch-and-bound decomposition approaches.

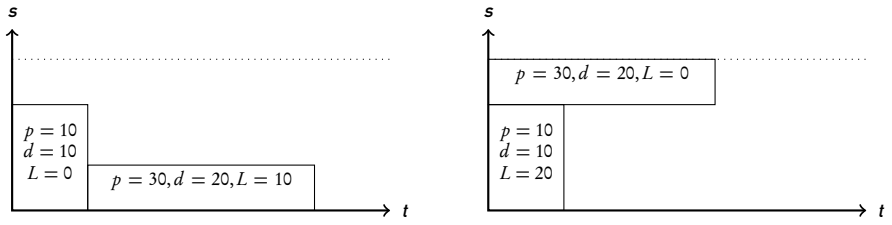


Figure 6.1: Overzealous batch elimination can increase $L_{\max}$

Unfortunately, we cannot make a general statement that optimal solutions never have more batches than other feasible solutions – a simple counterexample is shown in figure 6.1.[1]

Starting out with a one-job-per-batch schedule sorted by EDD, we can explore all feasible batch configurations recursively. To generate any other feasible schedule (including the optimal solution), jobs j are rescheduled ("moved back") from their original batch k_{origin} into a prior batch k_{earlier}. This eliminates k_{origin} and requires, of course, that k_{earlier} has sufficient capacity. If the job's processing time p_j exceeds that of k_{earlier}, then the lateness of batches between k_{earlier} and k_{origin} will increase; the merit of such a move cannot be judged a priori, so it is an *unsafe* batch elimination. *Safe* eliminations, on the other hand, will never worsen $L_{\max}$, and only they can be considered when bounding n_k a priori.

Algorithm 2 outlines a recursive method to find an upper bound on n_k, recognizing safe batch eliminations only.

This algorithm evidently requires exponential time. A relaxed variant is a possible option: if only a single unsafe move *into* a batch is possible, no safe eliminations into that batch are considered at all and we skip to the next batch. This would greatly speed up the recursion but also significantly weaken the usefulness of the resulting upper bound.

In a branch-and-bound decomposition approach in which batches are modelled individually, an upper bound on n_k could be used to limit the depth of the search tree, or, in

[1] To be more precise, we cannot state that at least one optimal solution is in the subset of feasible solutions that uses the fewest number of batches – a dominance situation that could be exploited, were it true.

Algorithm 2 Recursive algorithm to find an upper bound on n_k

```
if no open jobs left then                          ▷ if this is a "leaf node" in the recursion
    update UB(n_k)
end if
find the combination of unsafe later jobs that fills up the capacity most, leaving us with capacity b_r
find all combinations x of safe later jobs that fit into b_r
repeat
    x = next safe job combination
    ignoreJobs ← x                                 ▷ make the moves, let the next recursion level deal with the rest of the jobs
    spawn and run child node with J \ ignoreJobs
    ignoreJobs ← ignoreJobs \ x
until all combinations have been explored
return
```

combination with a method to determine a lower bound on the remaining jobs' n_k at every node, to actively prune the search tree during the search. The latter method, however, would also run in exponential time as it, again, would require knapsack-type reasoning unless we use a much less powerful relaxation.

6.3 CP Model

6.3.1 Constraint on number of batches with length $P_k > p$

Since batches take on the processing time of their longest job, there is at least one batch with $P = \max_j p_j$. We can proceed to fill batches with jobs, ordered by non-increasing processing time, based on algorithm 3.

At the end of this algorithm, we can state:

$$\texttt{globalCardinality}(\{1,\ldots,n_j\},\{P_{k,\min},\ldots,P_{k-1,\min}-1\},P_k) \quad \forall k \in \{k_0,\ldots,k_{LB(n_k)}\}, \tag{6.1}$$

where the constraint takes three arguments (`cards`, `vals` and `vars`) and $P_{k,\min}$ denotes the minimum length of batch k.

The algorithm sorts jobs by non-increasing p, and then fills batches job by job. If a job fits into a previous batch, it is assigned there. If a job fits into multiple previous batches, it is assigned to the batch with the smallest remaining capacity. This is called *best-fit decreasing* rule, and works as follows: let $J^\star$ be the set of jobs sorted by p, then at least one batch will

Algorithm 3 Generating lower bounds on batch lengths

```
J* ← J                                          ▷ initialize all jobs as unassigned jobs
n_k ← 1; S_k ← {0}; P_{k,min} ← {0}             ▷ Create one empty batch of size and length zero
sort J* by processing time, non-increasing
repeat
    j ← J*.pop()                                ▷ select job for assignment, longest job first
    loop through all n_k existing batches k, first batch first
        k_p ← ∅                                 ▷ no feasible batch
        c_min = b                               ▷ currently known minimum remaining capacity
        if s_j < b − S_k and b − S_k < c_min then
            k_p ← k_p; c_min ← b − S_k
        end if
    end loop
    if |k_p| = 1 then
        S_{k_p} ← S_{k_p} + s_j                 ▷ assign job j to batch k_p
    else
        if n_k < LB(n_k) then
            n_k ← n_k + 1                       ▷ open new batch
            S_{n_k} ← s_j; P_{n_k,min} ← p_j    ▷ assign s_j and p_j to the new batch
        else
            leave the loop now and end.
        end if
    end if
until J* is empty
```

be as long as the longest job $j_1^\star$. If the next n jobs fit into this batch, then there is at least one batch not shorter than $j_{n+1}^\star$, and similarly for subsequent batches.

Unfortunately, the optimal solution may perform better than the packing heuristic in terms of "vertical" (s_j) bin packing, and may thus require fewer batches. We therefore need to find a lower bound $LB(n_k)$ on the number of batches, and we can only guarantee the first $LB(n_k)$ of the above constraints to hold in the optimal solution. Finding a true lower bound is a two-dimensional bin packing problem, which is NP-hard. A possible but naive lower bound is j_0, the number of jobs, ordered by decreasing s_j, that can never fit into a batch together.

6.3.2 Constraint on number of batches with length $D_k > d$

In a similar fashion, we can determine that the second batch must be due no later than the earliest-due job j_{m+1} that can *not* fit into the first batch – if we sort jobs by due date and fit the earliest m jobs into the first batch – and so on for subsequent batches. Once again, since

best-fit decreasing may not perform optimally in terms of "vertical" packing, this may not be valid for batches beyond the known $LB(n_k)$.

6.3.3 All-different constraints on P_k and D_k

Furthermore, if all jobs have different processing times, all batches will have different processing times as well: `alldifferent`(P_k). If m out of n_j jobs have different processing times, we can still enforce `k_alldifferent`(P_k, m). Some work on `k_alldifferent` constraints has been done in [?].

Similarly, we knows that the constraint `k_alldifferent`(D_k, m) must be true if m out of n_j jobs have different due dates.

6.4 Decompositions, other approaches

6.4.1 Potential heuristics

Improve the initial $L_{\text{max,incumbent}}$ A better initial upper bound on $L_{\max}$ can help prune some branches of the search tree from the outset. There are several dispatch rules (or maybe other heuristics?) that could be explored to do this better.

Improve $L_{\text{max,incumbent}}$ during search It may be useful to use a heuristic like above to "complete the schedule" once a promising partial schedule has been generated. I have yet to identify situations where this is always helpful.

6.4.2 Move-back search

Another possible way to set up a branch-and-bound search for solutions works as follows: first, consider all jobs to be "single", i.e. assigned to individual batches such that $B_j = k_j \ \forall j \in J$. Compute the $L_{\max}$ for this schedule. Then, at every level of the search tree, move some single job j into any earlier batch $k \leqslant k_j$, but only if that move does not violate k's capacity.

If we start with a schedule of single jobs and only allow moving single jobs into earlier batches (and any schedule can be produced by a sequence of such moves!), we maintain the EDD ordering of batches. More importantly, such moves will never shift the position of $L_{\max}$ to the right:

In any partial schedule following EDD, let k be the batch with maximum lateness $L_k = L_{\max}$. It has processing time p_k. Then the lateness of the batch before k must be $L_{k-1} \geqslant L_k - p_k$ as a consequence of the EDD sequencing. Any batch following k can have a lateness no greater than $L_k - 1$.

- Moving any single job from a batch following k into a batch before k will worsen $L_{\max}$, but not change its position. Such a move is never necessary to arrive at an optimal solution.
- Moving back any single job from a batch before k *safely*[2] will improve $L_{\max}$, but not change its position.
- Moving back a single pre-k job j from a batch β into an earlier batch α *unsafely* will reduce L_k by p_α; $L_{\max}$ may still be in k, or it may be found in any batch between (and including) α and β, since their lateness $L_{[\alpha,\ldots,\beta]}$ is now increased by $p_j - p_\alpha$.
- If the max-lateness job j is single itself, it can be moved from k into an earlier batch α. If this is done *safely*, the batch immediately preceding k will still have a lateness $L_{k-1} \geqslant L_k - p_j$. All batches after k will have their lateness reduced by p_j, but since their maximum lateness did not exceed $L_k - 1$ before the move, it will now be at least 1 less than that of batch $k - 1$.
- If the max-lateness job j was moved back *unsafely* into a batch α, batches between and including α and $k - 1$ now have their lateness increased by $p_j - p_\alpha$, while batches after k have their lateness decreased by p_α. Again, batch L_{k-1} would exceed the maximum lateness of batches after k.

This shows that after a sequence of operations in which single jobs are moved into

[2] for the meaning of "safe" and "unsafe" moves, compare section 6.2.2.

earlier batches, $L_{\max}$ will never shift to the right. In fact, this means that all jobs after the $L_{\max}$-job in a single-job EDD schedule can be ignored in the scheduling problem entirely, although this turns out to be quite inconsequential as that job is often at or near the end of the single-job EDD schedule, resulting from the fact that $L_{k-1} \geqslant L_k - p_k$. By the same token, high-quality solutions often have their $L_{\max}$ in an early batch. This may give rise to some sort of decomposition method in which jobs are strategically moved back, and where we are trying to move $L_{\max}$ to the left as much as possible.

6.5 MIP model II

$$\text{Min.} \quad L_{\max} \tag{6.2}$$

$$\text{s.t.} \quad \sum_{k \in K} x_{jk} = 1 \qquad \forall j \in J \tag{6.3}$$

$$\sum_{j \in J} s_j x_{jk} \leqslant b \qquad \forall k \in K \tag{6.4}$$

$$x_{jk} = 0 \qquad \forall \{j \in J, k \in K | j \leqslant k \vee s_j + s_k > b \tag{6.5}$$

$$L_{\max} \geqslant \ell_k + \sum_{h=0}^{k-1} \left[P_h - p_h \left(1 + \sum_{i \in K} x_{hi} \right) \right] - \mathrm{UB}(L_{\max}) \sum_{i \in K} x_{ki} \qquad \forall k \in K \tag{6.6}$$

$$p_j x_{jk} \leqslant P_k \qquad \forall j \in J, \forall k \in K \tag{6.7}$$

$$\tag{6.8}$$

Model 6.5.1: Malapert's original MIP model

Bibliography

Meral Azizoglu and Scott Webster. Scheduling a batch processing machine with non-identical job sizes. *International Journal of Production Research*, 38(10):2173 – 2184, 2000.

Peter Brucker, Andrei Gladky, Han Hoogeveen, Mikhail Y. Kovalyov, Chris N. Potts, Thomas Tautenhahn, and Steef L. van de Velde. Scheduling a batching machine. *Journal of Scheduling*, 1(1):31–54, 1998.

Denis Daste, Christelle Gueret, and Chams Lahlou. A branch-and-price algorithm to minimize the maximum lateness on a batch processing machine. In *Proceedings of the 11th International Workshop on Project Management and Scheduling PMS'08, Istanbul, Turkey*, pages 64–69, 2008a.

Denis Daste, Christelle Gueret, and Chams Lahlou. Génération de colonnes pour lordonnancement dune machine à traitement par fournées. In *7ème conférence internationale de modélisation et simulation (MOSIM 2008), Paris, France*, volume 3, pages 64–69, 2008b.

Lionel Dupont and Clarisse Dhaenens-Flipo. Minimizing the makespan on a batch machine with non-identical job sizes: an exact procedure. *Computers & Operations Research*, 29(7):807 – 819, 2002.

Ignacio E. Grossmann. Mixed-integer optimization techniques for the design and scheduling of batch processes. Technical Report Paper 203, Carnegie Mellon University Engineering Design Research Center and Department of Chemical Engineering, 1992.

Ali Husseinzadeh Kashan, Behrooz Karimi, and S. M. T. Fatemi Ghomi. A note on minimizing makespan on a single batch processing machine with nonidentical job sizes. *Theoretical Computer Science*, 410(27-29):2754–2758, 2009.

Arnaud Malapert, Christelle Guéret, and Louis-Martin Rousseau. A constraint programming approach for a batch processing problem with non-identical job sizes. *European Journal of Operational Research*, 221:533–545, 2012.

Onur Ozturk, Marie-Laure Espinouse, Maria Di Mascolo, and Alexia Gouin. Makespan minimisation on parallel batch processing machines with non-identical job sizes and release dates. *International Journal of Production Research*, 50(20):6022–6035, 2012.

M.T. Yazdani Sabouni and F. Jolai. Optimal methods for batch processing problem with makespan and maximum lateness objectives. *Applied Mathematical Modelling*, 34(2):314 – 324, 2010.

Appendix: Tables

A.1 Hello. This is the first part of the first Appendix

Blabla.

Appendix: Code